PandA: Pairings and Arithmetic

Chitchanok Chuengsatiansup[1], Michael Naehrig[2], Pance Ribarski[3], and Peter Schwabe[4] ⋆

[1] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, the Netherlands
c.chuengsatiansup@tue.nl
[2] Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
michael@cryptosith.org
[3] Faculty of Computer Science and Engineering
Ss. Cyril and Methodius University in Skopje
P.O. Box 393, 1000 Skopje, Republic of Macedonia
pance.ribarski@finki.ukim.mk
[4] Digital Security Group
Radboud University Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
peter@cryptojedi.org

Abstract. This paper introduces PandA, a software framework for *Pairings and Arithmetic*. It is designed to bring together advances in the efficient computation of cryptographic pairings and the development and implementation of pairing-based protocols. The intention behind the PandA framework is to give protocol designers and implementors easy access to a toolbox of all functions needed for implementing pairing-based cryptographic protocols, while making it possible to use state-of-the-art algorithms for pairing computation and group arithmetic. PandA offers an API in the C programming language and all arithmetic operations run in constant time to protect against timing attacks. The framework also makes it easy to consistently test and benchmark the lower level functions used in pairing-based protocols.
As an example of how easy it is to implement pairing-based protocols with PandA, we use Boneh-Lynn-Shacham (BLS) signatures. Our PandA-based implementation of BLS needs only 434640 cycles for signature generation and 5832584 cycles for signature verification on one core of an Intel i5-3210M CPU. This includes full protection against timing attacks and compression of public keys and signatures.

Keywords: Cryptographic pairings, benchmarking, API design, BLS signatures

1 Introduction

Since the late 1990s and early 2000s, when Ohgishi, Sakai, Kasahara [46,51,52] and Joux [39,40] presented the first constructive uses of cryptographic pairings, many pairing-based cryptographic protocols have been proposed. Early work such as the identity-based encryption scheme by Boneh and Franklin [18] and the short signature scheme by Boneh, Lynn and Shacham [19], were followed by a flood of papers presenting more and more pairing-based schemes with exciting, new cryptographic functionalities. Examples include schemes for hierarchical identity-based encryption [38,29], attribute-based encryption [50], systems for non-interactive zero-knowledge proofs [35,34], and randomizable proofs and anonymous credentials [10].

In a highly related—but often somewhat independent—line of research, the performance of pairing computation was drastically improved. Milestones in this line of research were the construction of various families of pairing-friendly curves (for an overview, see [28]), many optimizations for the pairing algorithm including denominator elimination in the Miller loop [7],

⋆ This work was supported by the European Commission under ICT COST Action IC1204 TRUDEVICE. Permanent ID of this document: 775a51985db9972bde7bd2acddf1d2a2. Date: December 4, 2013

faster algorithms to compute the final exponentiation [56], and the introduction of loop-shortening techniques [36], that lead to the notion of *optimal pairings* [61]. Recently, several papers presented high-speed software that computes 128-bit secure pairings for various Intel and AMD processors [45,17,5,43], and for ARM processors with NEON support [53]. These efforts reduced the time required to compute a pairing at the 128-bit security level on current processors to below 0.5 ms.

Unfortunately, these advances in pairing performance do not immediately speed up pairing-based protocols. The reason is that protocols need much more than just fast pairings. They need fast arithmetic in all involved groups, fast hashing into elliptic-curve groups, fast multi-scalar multiplication (and multi-exponentiation), or specific optimizations for computing products of pairings. This means that, even if authors of speed-record papers for pairing computation make their software available, this software is typically not "complete" from a protocol designer's point of view, and does not necessarily include these other operations; and it is often not easy to use when it comes to prototyping a new pairing-based protocol to evaluate its practical performance. Also, once a protocol implementation has settled for one pairing library, it typically requires a significant effort to switch to another software or library.

Furthermore, as Scott points out in [54], which optimizations to the pairing computation or other arithmetic operations are most useful, strongly depends on the pairing-based protocol that is being implemented. Pairings are used in such protocols in different flavors, where in some scenarios pairing computation is the dominant cost in the overall protocol and in others the large number of non-pairing operations may be the bottleneck (see, for example, [48]). If the protocol contains many more group exponentiations than it has pairing computations, in some cases it might even make sense to choose different pairing-friendly curves to allow faster group operations at the cost of a slightly more expensive pairing (see the ratios of group exponentiation and pairing costs in [20]). In an implementation that has been tailored for high-speed pairings only, it is often difficult to account for such trade-offs.

This paper introduces PandA, a software framework that intends to address the above concerns by making improvements in pairing (and more generally group-arithmetic) performance easily usable for protocol designers. The project is inspired by the eBACS benchmarking project [14] that defines APIs for various typical cryptographic primitives and protocols (such as hash functions, stream ciphers, public-key encryption, and cryptographic signatures). PandA can be seen as a generalization of eBACS to lower-level functions in the elliptic-curve and pairing setting. We are currently discussing a possible inclusion of PandA into eBACS with the editors of the eBACS project.

We encourage submissions of implementations of all the underlying functions to extend the implementation portfolio and to obtain consistent benchmarking as shown in the eBACS project. In particular, we hope that implementors of pairings will be motivated to submit more complete libraries that allow the implementation of full pairing-based protocols. We will make all software described in this paper available at `http://panda.cryptojedi.org` and place it in the public domain to maximize reusability of our results.

Type-1, Type-2 and Type-3 pairings. Currently our reference implementation of the PandA API only implements a particular set of parameters for Type-3 pairings, but the API is designed to support arbitrary pairing-friendly curves. However, Section 2 explains how the API supports also Type-1 pairings. Until recently the standard approach to implementing high-security (e.g., 128-bit secure) Type-1 pairings was using supersingular curves over binary or ternary fields. However, advances on solving discrete-logarithm problems in multiplicative groups of small-characteristic fields by Joux in [41], by Göloğlu, Granger, McGuire, and Zumbrägel in [31], by Barbulescu, Gaudry, Joux, and Thomé in [6], and by Adj, Menezes, Oliveira, and Rodríguez-Henríquez in [1] have raised serious concerns about the security of such constructions. Granger commented that he does not "think the coffin has been firmly

nailed shut just yet!" [5], and it is indeed not clear that all small-characteristic pairings are broken, but there is a strong consensus that pairings on curves over small-characteristic fields are not recommended anymore. We are therefore planning to include a reference implementation of Type-1 pairings that uses an approach similar to the ones described in [59] and [63].

We follow Chatterjee and Menezes stating in [24] that "Type 2 pairings are merely inefficient implementations of Type 3 pairings, and appear to offer no benefit for protocols based on asymmetric pairings from the point of view of functionality, security, and performance". Thus, we do not explicitly support Type-2 pairings, but it would be straight-forward to include Type-2 pairings in PandA (the only difference from an API perspective is missing hashing into the second group of pairing arguments).

The Type-3 pairing setting in this paper is as follows. The pairing is a non-degenerate, bilinear function $e : G_1 \times G_2 \to G_3$, where G_1 and G_2 are groups of prime order r consisting of rational points on an ordinary, pairing-friendly elliptic curve E defined over a finite field $\mathbb{F}_p$ of prime characteristic p. The elliptic curve E has a small embedding degree k, which means that the group G_3 is the group of r-th roots of unity in the multiplicative group $\mathbb{F}^*_{p^k}$, i.e. all three groups have prime order r.

Arithmetic in non-pairing groups. PandA also has an API for arithmetic in groups that do not support efficient computation of pairings (like non-pairing-friendly elliptic curves). If protocols do not need efficient pairing computation they can choose from a much larger pool of groups in which the DLP is hard. When choosing from this larger pool one can typically pick groups with more efficient arithmetic. The group API supports all functions that are also supported for each of the three groups in the pairing setting. Our reference implementation of this API uses the group of the twisted Edwards curve that is also used for Ed25519 signatures [12,13]. However, this paper focuses on the description of the pairing setting in PandA.

The importance of constant-time algorithms. Aside from attacks against the hard problems that the security of modern cryptography is based on, major threats to cryptographic software are side-channel attacks. In particular timing attacks (that can even be carried out remotely in many cases) prove to be a very powerful attack tool. See [47,60], [22], [26], [62] for some examples of timing attacks against cryptographic software.

One could argue that a framework which is designed to evaluate the performance of cryptographic protocols should not pay attention to these issues, but rather keep the API simple, and add suitable timing-attack protection only for "real-world" software. We disagree for two reasons. First, once some pieces of unprotected cryptographic software have been written and publicized, it is almost impossible to ensure that it does not end up in some real-world software. Second, and more importantly, protecting software against timing-attacks does not add a constant overhead; the cost highly depends on protocol design, and algorithm and parameter choices made on a high level. For example, the completeness of the group law on Edwards curves [15,11] makes it easy to protect group addition against timing attacks. It is possible to protect Weierstrass-curve point addition against timing attacks (see Section 3) but it involves a significant overhead.

Optimizing performance of unprotected implementations of cryptographic protocols may thus lead to wrong decisions that are very hard to correct later. PandA acknowledges this fact by offering timing-attack protected (constant-time) versions of all arithmetic operations. For operations that do not involve any secret data (such as signature verification) there are possibly faster non-constant-time versions of all group-arithmetic operations. These unprotected versions of functions have to be chosen explicitly; the default is the constant-time versions.

[5] see http://ellipticnews.wordpress.com/2013/05/22/joux-kills-pairings-in-characteristic-2/

Related work. There exist various cryptographic *libraries* that expose low-level functionality such as group arithmetic and pairings through their API. However, the API that gives access to this low-level functionality is typically tailored to suit the specific needs of the higher-level primitives of the library. It is usually not designed for efficient implementation of arbitrary new protocols. Some libraries that use group arithmetic even decide to not expose the low-level functionality through the API, because this functionality was never written to be used outside the specific needs of the higher-level protocols. See, for example, the high-level API of NaCl [16]. Two notable examples of cryptographic libraries with a convenient API for pairings and group arithmetic are RELIC [4] and Miracl [23].

A library which has been explicitly designed for the use in arbitrary pairing-based protocols is the PBC library [42]. This careful design is the reason that it is still the preferred library for the implementation of various protocols; despite the fact that it does not offer state-of-the-art performance and (by default) no high-security curves. The PandA API is designed with the same usage profile as PBC in mind. However, the reference implementation of the PandA API presented in this paper offers state-of-the art performance with a curve choice that offers 128 bits of security. Furthermore, PandA is designed as a framework that supports (and encourages!) submissions by various designers to keep reflecting the state-of-the-art in group-arithmetic and pairing performance.

Another framework for easy implementation of cryptographic protocols is Charm [3]. Charm offers a high-level Python API and uses multiple cryptographic libraries to achieve good performance. For pairing-based cryptography it uses the PBC library. Charm is a higher-level framework than PandA; we see PandA not in competition to Charm but rather hope that Charm will eventually include some of PandA's high-performance pairing and group-arithmetic implementations to speed up protocols implemented in its high-level API.

Organization of the paper. Section 2 explains the PandA API. Section 3 gives details of our reference implementation of this API and reports benchmark results of all arithmetic operations. Section 4 considers an example that shows how easy it is to implement pairing-based protocols that achieve state-of-the-art performance using the PandA API.

2 PandA API and functionality

The API of PandA is inspired by the API of eBACS, which means in particular that the API is also for the C programming language. There are various advantages of using C. It is the language most commonly used for speed-record-setting cryptographic software (often combined with assembly), so a C API makes it easy to integrate fast software in PandA. Furthermore, protocols that need group arithmetic, pairings, and, for example, a hash function or a stream cipher, can easily combine software from PandA with software that is tested and benchmarked in eBACS.

In the eBACS API all functions are within the `crypto` namespace, i.e. all function names begin with `crypto_`. Similarly, all functions and data types related to arithmetic in groups that support efficient bilinear-pairing computation are in the `bgroup` namespace (for "bilinear group"); the API for group arithmetic without pairings uses the `group` namespace.

2.1 PandA data types

The functionality that is tested and benchmarked in PandA is on a lower level in the design of cryptographic protocols. In the eBACS project, complete cryptographic primitives and protocols are benchmarked, while PandA benchmarks arithmetic operations that are meant to be used to implement cryptographic protocols. This has consequences for the data type of inputs and outputs. In eBACS, all functions receive inputs as byte arrays (C data type `unsigned`

`char`), the length of these arrays is specified in arguments of type `unsigned long long`. Outputs are again written to byte arrays. A typical implementation of a cryptographic protocol in eBACS first converts the input byte arrays to an internal representation for fast computation that depends on the architecture, then performs all computations in this representation, and then transforms the output to a unique representation as a byte array. These transformations typically contribute only little overhead to the cost of a cryptographic protocol if they are done only at the beginning and the end of the *protocol*. Protocols implemented using the PandA API typically need a sequence of functions from the PandA API and we clearly want to avoid transformations at the beginning and the end of each *function*. Implementations of the PandA API therefore define 4 data types—for elements of the three groups G_1, G_2 and G_3 and for scalars (modulo the group order)—in a file called `api.h`. These data types (`struct` in C) are called `bgroup_g1e`, `bgroup_g2e`, `bgroup_g3e`, and `bgroup_scalar`. The API provides two functions, one is used to convert an element of G_1, G_2, G_3, or a scalar to a unique byte array of fixed length (`pack`), the other one converts such a byte array back to a group element or scalar (`unpack`). Implementations of the PandA API furthermore specify the size of packed elements in `api.h`:

```
#define BGROUP_G1E_PACKEDBYTES 32
#define BGROUP_G2E_PACKEDBYTES 64
#define BGROUP_G3E_PACKEDBYTES 384
#define BGROUP_SCALAR_PACKEDBYTES 32
```

indicating that packed elements of G_1 need 32 bytes, packed elements of G_2 need 64 bytes, etc. From this file, PandA automatically generates the header file `panda_bgroup.h` that defines all functions of the API. Implementations of Type-1 pairings omit the implementation of G_2 and instead include a line

```
#define BGROUP_TYPE1
```

in the file `api.h`. For the group G_1, the `unpack` and `pack` functions are

```
int bgroup_g1e_unpack(
          bgroup_g1e *r,
          const unsigned char b[BGROUP_G1E_PACKEDBYTES]);

void bgroup_g1e_pack(
          unsigned char r[BGROUP_G1E_PACKEDBYTES],
          const bgroup_g1e *b);
```

Following eBACS convention, the `unpack` function returns an integer value, which is zero whenever a valid byte array is received that can be unpacked to a group element. On input of an invalid byte array that does not correspond to a packed group element, the function returns a non-zero integer. In the following, we mostly describe the API for arithmetic in G_1 as an example. Equivalent functions exist for G_2 and G_3.

2.2 PandA constants

For each of the three groups, a PandA implementation has to define two constants: a generator and the neutral element. For the group G_1 these are called `bgroup_g1e_base` and `bgroup_g1e_neutral`. Each implementation needs to ensure that the pairing evaluated at `bgroup_g1e_base` and `bgroup_g2e_base` gives `bgroup_g3e_base` as result. Furthermore, each PandA implementation has to define two constants of type `bgroup_scalar` for the element zero and the element one in the ring of integers modulo the order r of the groups G_1, G_2, and G_3. These constants are called `bgroup_scalar_zero` and `bgroup_scalar_one`.

2.3 Comparing group elements

One way to compare two group elements for equality is obviously to use the `bgroup_g1e_pack` function on both of them and compare the resulting byte arrays for equality. This is typically not the most efficient way to compare equality (except if packing of elements is required anyway). For example, consider two elliptic-curve points in projective coordinates. Conversion to a *unique* byte array requires transformation to affine coordinates, i.e., two inversions and several multiplications. Comparison for equality only needs a few multiplications. The API therefore has a comparison function

```
int bgroup_g1e_equals(const bgroup_g1e *a, const bgroup_g1e *b);
```

which returns 1 if the two elements are equal and 0 if they are not.

As explained in the introduction, this function must be guaranteed to not leak timing information about the two arguments. For cases where none of the two inputs is secret, there is a function

```
int bgroup_g1e_equals_publicinputs(const bgroup_g1e *a, const bgroup_g1e *b);
```

which behaves the same way but is not guaranteed to not leak timing information and may be faster than the constant-time version.

2.4 Addition and doubling

In concrete implementations of pairings, the groups G_1 and G_2 are typically additive groups, while the group G_3 is a multiplicative group. Hence, the core operations for group arithmetic are additions and doublings in G_1 and G_2 and multiplications and squarings in G_3. It makes sense to treat all three groups as abstract abelian groups and therefore use a common notation for the group operation in all of them. Many papers that treat a pairing as a black box use multiplicative notation for G_1, G_2, and G_3. Instead, the PandA API uses additive notation following the `crypto_scalarmult` API of the SUPERCOP benchmarking framework used in eBACS.

Addition of two elements, doubling, and negation (computing the inverse of an element) are done through the functions:

```
void bgroup_g1e_add(bgroup_g1e *r, const bgroup_g1e *a, const bgroup_g1e *b);
void bgroup_g1e_double(bgroup_g1e *r, const bgroup_g1e *a);
void bgroup_g1e_negate(bgroup_g1e *r, const bgroup_g1e *a);
```

Note that the return value is always written to the first argument pointer (as in the eBACS API and also, for example, in the GMP API [30]). Note also that the implementation needs to ensure that the addition and doubling functions work for all elements of the group as inputs and that no timing information leaks about these inputs or the output. As before, there are also potentially faster non-constant-time versions of these functions:

```
void bgroup_g1e_add_publicinputs(bgroup_g1e *r, const bgroup_g1e *a, const bgroup_g1e *b);
void bgroup_g1e_double_publicinputs(bgroup_g1e *r, const bgroup_g1e *a);
void bgroup_g1e_negate_publicinputs(bgroup_g1e *r, const bgroup_g1e *a);
```

2.5 Scalar multiplication

The default function for performing a scalar multiplication is simply

```
void bgroup_g1e_scalarmult(bgroup_g1e *r, const bgroup_g1e *a, const bgroup_scalar *s);
```

This function can be made much faster when multiplying a fixed base point that is known at compile time. This potentially faster version is supported for the generator `bgroup_g1e_base` through

```
void bgroup_g1e_scalarmult_base(bgroup_g1e *r, const bgroup_scalar *s);
```

Another improvement can be implemented for multi-scalar multiplication, i.e. whenever a sum $\sum_{i=0}^{m-1} s_i P_i$ of several scalar multiples needs to be computed for m scalars $s_0, \ldots, s_{m-1}$ and m group elements $P_0, \ldots, P_{m-1}$. Such computations are supported through the below function, in which the last (`unsigned long long`) argument specifies the number m of scalar multiplications to be performed in the sum.

```
void bgroup_g1e_multiscalarmult(
                bgroup_g1e *r, const bgroup_g1e *a,
                const bgroup_scalar *s, unsigned long long alen);
```

Again, all group elements have to be supported as inputs, constant-time behavior has to be ensured by implementations, and the API also supports non-constant-time (`publicinputs`) versions of the functions. The input `alen` is considered public also for the constant-time version.

2.6 Hashing to G_1 and G_2

Many protocols require hashing of arbitrary bit strings to group elements in G_1 and G_2, which is also supported by the PandA API. The corresponding function for hashing into G_1 is:

```
void bgroup_g1e_hashfromstr(bgroup_g1e *r, const unsigned char *a, unsigned long long alen);
```

As for the previous functions, there is also a non-constant-time (`publicinputs`) version of this function. Due to the different ways in which the constant-time and non-constant-time functions are computed, it can be the case that the hashed values obtained by evaluating each version on the same input bit string are different. It is not necessary to insist that both versions compute the same result, because we expect that throughout a protocol, the same input string to a hash function is always either public or private. Therefore, one can consistently select the right version of the function and thus take advantage of faster non-constant-time algorithms.

2.7 Arithmetic on scalars

Various functions are supported for arithmetic on scalars modulo the group order, which are required in various protocols (for example for ECDSA signatures). Specifically these functions are the following:

```
void bgroup_scalar_setrandom(bgroup_scalar *r);
void bgroup_scalar_add(bgroup_scalar *r, const bgroup_scalar *s, const bgroup_scalar *t);
void bgroup_scalar_sub(bgroup_scalar *r, const bgroup_scalar *s, const bgroup_scalar *t);
void bgroup_scalar_negate(bgroup_scalar *r, const bgroup_scalar *s);
void bgroup_scalar_mul(bgroup_scalar *r, const bgroup_scalar *s, const bgroup_scalar *t);
void bgroup_scalar_square(bgroup_scalar *r, const bgroup_scalar *s);
void bgroup_scalar_invert(bgroup_scalar *r, const bgroup_scalar *s);
int  bgroup_scalar_equals(const bgroup_scalar *s,  const bgroup_scalar *t);
```

Arithmetic on scalars is typically not the performance bottleneck in pairing-based protocols; furthermore we do not expect significant speedups for non-constant-time versions of scalar arithmetic. Therefore, the API does not include `publicinputs` versions of functions for arithmetic on scalars.

2.8 Pairings and products of pairings

Finally, the API function for computing a pairing is

```
void bgroup_pairing(bgroup_g3e *r, const bgroup_g1e *a, const bgroup_g2e *b);
```

Some protocols need—or can make use of—the product of several pairings (for an example see BLS signatures in Section 4). Computing the product of two pairings can be significantly faster than computing two independent pairings and then multiplying the results. One reason is that the final exponentiation has do be done only once, another reason is that squarings inside the Miller loop can be shared between the two pairings. To support these important speedups, the PandA API includes a function

```
void bgroup_pairing_product(
            bgroup_g3e *r, const bgroup_g1e *a, const bgroup_g2e *b,
            unsigned long long alen);
```

3 PandA reference implementation

This section describes our reference implementation of the API functions from Section 2. The implementation provides a 128-bit secure, Type-3 pairing framework.

3.1 Choice of parameters

At the 128-bit security level, the most suitable choice of pairing-friendly curve is a Barreto-Naehrig curve [8] over a prime field of size roughly 256 bits. We use the 254-bit curve $E = E_{2,254}$ that has been proposed in [49] and has also been used in [5]. The curve parameter $u = -(2^{62} + 2^{55} + 1)$ yields 254-bit primes $p = p(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1$ and $r = r(u) = 36u^4 + 36u^3 + 18u^2 + 6u + 1$, and $E : y^2 = x^3 + 2$ over $\mathbb{F}_p$. Since the embedding degree is $k = 12$, the implementation needs to provide the field extension $\mathbb{F}_{p^{12}}$. This extension is implemented in the standard way as a tower $\mathbb{F}_p \subset \mathbb{F}_{p^2} \subset \mathbb{F}_{p^6} \subset \mathbb{F}_{p^{12}}$.

As usual, the elliptic-curve groups are $G_1 = E(\mathbb{F}_p)$ and G_2 is the p-eigenspace of the p-power Frobenius in the r-torsion group $E(\mathbb{F}_{p^{12}})[r]$, which is represented by an isomorphic group $G_2' = E'(\mathbb{F}_{p^2})[r]$ on a sextic twist E' of E over $\mathbb{F}_{p^{12}}$. Whenever we work with elements in G_2, we make use of their representation as elements in G_2', i.e. they are curve points with coefficients in $\mathbb{F}_{p^2}$ and arithmetic is actually arithmetic on E' over $\mathbb{F}_{p^2}$.

3.2 Algorithms

Packing and unpacking. To pack elements of the groups G_1 and G_2, we use the usual way of point compression on elliptic curves. For elliptic-curve arithmetic, points are in Jacobian coordinates. To pack a point, it is first transformed to affine coordinates. The packed representation is the 32-byte array containing the point's 254-bit affine x-coordinate together with the least significant bit of its y-coordinate in one of the remaining two free bits. The other free bit is used to represent the point at infinity.

Given such a byte array, the unpacking algorithm recovers the x-coordinate and solves the curve equation for the y-coordinate, choosing the right square root according to the least significant bit given in the array. The core of this operation is a square root computation, for which we use different algorithms in G_1 and G_2. Since $p \equiv 3 \mod 4$, in G_1, we use $a^{(p+1)/4}$ to compute the square root of $a \in \mathbb{F}_p$. The unpack algorithm in G_2 uses [2, Algorithm 9] to compute the square root. After obtaining a point on the curve, it needs to be checked whether it has order r, i.e. whether it is in the correct subgroup.

The elements of G_3 are kept as elements of $\mathbb{F}^*_{p^{12}}$. The packing algorithm constructs a unique byte array composed of the twelve $\mathbb{F}_p$-coefficients of the unique $\mathbb{F}_{p^{12}}$-element in G_3. The unpack algorithm simply converts the byte array back to an $\mathbb{F}_{p^{12}}$-element and checks that the order of the element is r. At the time of writing this paper, the implementation does not compress pairing values. However, this will be changed. Pairing values can be compressed to one third the length of an $\mathbb{F}_{p^{12}}$-element by using the techniques described in [55,32,44,5].

Comparison. To compare elements of the groups G_1 and G_2, we need to compare points that are represented in (projective) Jacobian coordinates. The standard way of comparing these redundant representations is to multiply through by the respective powers of the Z-coordinate. This does not need inversions, in contrast to a conversion to affine coordinates. Comparison in the group G_3 can directly compare $\mathbb{F}_{p^{12}}$-elements or the respective compressed representations.

Hashing to G_1 and G_2. The standard non-constant-time algorithm to hash an arbitrary string to a point on an elliptic curve is the "try-and-increment" method introduced in [19]. The message is concatenated with a counter and hashed by a cryptographic hash function to an element of the underlying finite field. If this element is a valid x-coordinate, compute one of the corresponding y coordinates; otherwise increase the counter and repeat the procedure. We use this method for non-constant-time hashing to G_1 and G_2.

For constant-time hashing to G_1 and G_2 we use the algorithm described in [27] which is based on the algorithm by Shallue and van der Woestijne described in [57]. The conditional branches in the algorithm (in particular choosing between one out of three possible solutions) are implemented through constant-time conditional-copy operations.

We do not yet include the indistinguishable hashing described in [27]. This would require carrying out two independent hashing operations to the curve (e.g., by using two different cryptographic hash functions) and then adding the results.

Group addition. We represent elements of G_1 and G_2 in Jacobian coordinates. For non-constant-time addition we use the addition formulas by Bernstein and Lange that take 11 multiplications and 5 squarings[6]. If the inputs happen to be one of the special cases that are not handled by the formulas we use conditional branches to switch to doubling or to returning the point at infinity. Doubling uses the formulas by Lange that take 5 squarings and 2 multiplications[7].

Constant-time complete addition on a Weierstrass curve is not easy to do efficiently. There exist no complete formulas [21, Theorem 1]. The unified formulas proposed in [37, 5.5.2] can handle doublings but they achieve this by moving the special cases to other points (specifically, addition of points of the form (x_1, y_1) and $(x_2, -y_1)$ with $x_1 \neq x_2$). Here, we evaluate two sets of formulas and use constant-time conditional copies to choose between the two outputs. We do that with the addition and doubling formulas described above. Note that protocols are typically not bottlenecked by additions but rather by scalar multiplications. Constant-time scalar-multiplication can use much faster dedicated addition as long as we can be sure that scalars are smaller than the group order. This is also compatible with the GLV/GLS decomposition described in the next paragraph.

Scalar multiplication. For the scalar multiplication algorithms implemented in PandA for each of the three groups, we distinguish between constant-time algorithms and their more efficient counterparts public inputs. For each case, there are three algorithms: general scalar multiplication, scalar multiplication of a fixed base point, and multi-scalar multiplication.

Scalar multiplication of a fixed base point that is known at compile time is done by precomputing 512 multiples of that point in a table and then using these to compute the scalar multiple. The method we use is described in detail by Bernstein et al. [12,13, Section 4]. Since we do not expect a significant speed-up by moving from the constant-time to a variable-time version, we also use the constant-time algorithm in the function on public inputs.

The standard case of scalar multiplication uses efficient endomorphisms on the BN curve by splitting the scalars via 2-dimensional GLV in G_1 and 4-dimensional GLS decomposition in G_2 and G_3. See the work by Bos, Costello, and Naehrig [20] for details. In G_1, we slightly

[6] http://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#addition-add-2007-bl
[7] http://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#doubling-dbl-2009-l

differ from the method in [20]. After the scalar decomposition in the constant-time function, we save a few additions by using a fixed signed window of size 5 and two additions per lookup, instead of the table with window size 2 and one addition. The function on public inputs uses a signed sliding window of size 5. The constant-time algorithms in G_2 and G_3 are as described in [20], the variable-time algorithms use signed sliding windows of size 4.

The variable-time algorithm for multi-scalar multiplication first applies the GLV/GLS scalar decomposition. For small batch sizes it then uses joint-signed-sliding-window scalar multiplication; for larger batch sizes (> 16 for G_1 and > 8 for G_2 and G_3) we use Bos-Coster scalar multiplication (described in [25, Section 4]). For the constant-time version, due to the slow complete addition routine, the function currently simply carries out each scalar multiplication separately and adds them together at the end. For the group G_3, it seems worthwhile to implement exponentiations of compressed values using the methods of Stam and Lenstra [58]. We are planning to consider this optimization.

Pairing computation. The pairing algorithm computes the optimal ate pairing on the same BN curve as [5]. Unlike [5] we do not use standard projective coordinates but Jacobian coordinates as in [45]. We use lazy reduction for arithmetic in the extension fields as described in [5]. The final exponentiation implements the same approach as [17], we use the cyclotomic squarings from [33, Section 3.1], but we do not use the compressed squarings described in [5, Section 5.2].

We are planning to continue optimizing pairing computation by experimenting with standard projective coordinates, a final exponentiation with compressed squarings [5], and faster low-level arithmetic.

Low-level arithmetic. The low-level arithmetic in $\mathbb{F}_p$ and arithmetic on scalars are implemented in `AMD64` assembly. We use Montgomery representation for elements in $\mathbb{F}_p$; scalars are represented in "standard" form because in scalar multiplication we need access to the binary representation. Modular reduction of scalars uses Barrett reduction [9].

We have not yet implemented inlined arithmetic in $\mathbb{F}_{p^2}$ in assembly. We are planning to include this optimization and expect significant performance improvements for pairing computation and for arithmetic in G_2 and G_3.

We also have a compatible implementation of the field arithmetic entirely written in C to support other platforms. We will continue to optimize the software with assembly implementations for other platforms, in particular ARM processors with NEON support.

3.3 Performance

We benchmarked our software (with $\mathbb{F}_p$ arithmetic implemented in assembly) on one core of an Intel Core i5-3210M processor with Turbo Boost and Hyperthreading disabled. For each function we carried out 100 computations on random inputs. The median and quartiles of the cycle counts measured in these experiments are reported in Tables 1, 2, 3, and 4.

4 Implementing protocols with PandA

In this section we consider BLS signatures [19] as a small example of a pairing-based protocol implemented in PandA. We choose this example, because it illustrates the use of most API functions of PandA and because cryptographic signatures (unlike more complex cryptographic protocols) are supported by the eBACS benchmarking project [14]. The software presented in this section implements the eBACS API for cryptographic signatures and we will submit our software to eBACS for public benchmarking.

API function	25% quartile	median	75% quartile
bgroup_g1e_unpack	39140	39184	39212
bgroup_g1e_pack	39512	39548	39568
bgroup_g1e_hashfromstr (59 bytes)	198780	198908	198964
bgroup_g1e_add	6052	6080	6100
bgroup_g1e_double	1204	1216	1224
bgroup_g1e_negate	36	36	40
bgroup_g1e_scalarmult	346852	347024	347180
bgroup_g1e_scalarmult_base	128468	128596	128696
bgroup_g1e_multiscalarmult			
($n = 2$)	705564	705820	706056
($n = 3$)	1058308	1058644	1059128
($n = 4$)	1411188	1411644	1411944
($n = 8$)	2822252	2823148	2826864
($n = 32$)	11294736	11296364	11298420
($n = 128$)	45181816	45186732	45193356
bgroup_g1e_equals	1124	1132	1140
bgroup_g1e_hashfromstr_publicinputs (59 bytes)	41752	83168	83696
bgroup_g1e_add_publicinputs	2456	2468	2476
bgroup_g1e_double_publicinputs	1180	1192	1200
bgroup_g1e_negate_publicinputs	36	36	40
bgroup_g1e_scalarmult_publicinputs	284228	288240	290788
bgroup_g1e_scalarmult_base_publicinputs	102184	104024	105772
bgroup_g1e_multiscalarmult_publicinputs			
($n = 2$)	415076	419860	423440
($n = 3$)	551124	556792	560712
($n = 4$)	710416	715396	722000
($n = 8$)	1229100	1238660	1246568
($n = 32$)	4727808	4741472	4752772
($n = 128$)	14590168	14605364	14635184
bgroup_g1e_equals_publicinputs	576	580	588

Table 1. Cycle counts for arithmetic operations in G_1 on Intel Core i5-3210M.

4.1 The BLS signature scheme

We briefly describe the three algorithms — key generation, signing, and verification — of the BLS scheme for an asymmetric, Type-3 pairing. Let $Q \in G_2$ be a system-wide fixed base point for G_2.

Key generation. Pick a random scalar $s \in \mathbb{Z}_r^*$. Compute the scalar multiple $R \leftarrow [s]Q$. Return R as the public key and s as the private key.

Signing. Hash the message m to an element M in G_1. Use the private key s to compute $S = [s]M$. Return the x-coordinate of the result S as the signature σ.

Verification. Upon receiving a signature σ, a message m, and the public key R, find an element $S \in G_1$ such that its x-coordinate corresponds to σ and it has order r. If no such point exists, reject the signature. Then calculate $t_1 \leftarrow e(S, Q)$. Compute the hash $M \in G_1$ of the message m, and compute $t_2 \leftarrow e(M, R)$. The signature is accepted if $t_1 = t_2$ or $t_1 = -t_2$ and rejected otherwise. Note that we use additive notation in G_3.

This scheme requires one scalar multiplication for key generation, one scalar multiplication for signature generation, and the comparison of two pairing values for signature verification. In our case the signature is the packed value of the elliptic-curve point, which includes the information on the sign of the correct y-coordinate. We therefore compute the unique point S corresponding to the signature σ, and to verify we only need to check whether $e(-S, Q) \cdot e(M, R) = 1$.

API function	25% quartile	median	75% quartile
bgroup_g2e_unpack	1864580	1864884	1865396
bgroup_g2e_pack	42080	42124	42160
bgroup_g2e_hashfromstr (59 bytes)	2435116	2435564	2439536
bgroup_g2e_add	16048	16072	16096
bgroup_g2e_double	2924	2940	2948
bgroup_g2e_negate	60	60	64
bgroup_g2e_scalarmult	764628	764808	765088
bgroup_g2e_scalarmult_base	336788	336916	337060
bgroup_g2e_multiscalarmult			
($n = 2$)	1563312	1563668	1564040
($n = 3$)	2344964	2345496	2346704
($n = 4$)	3126720	3127116	3131192
($n = 8$)	6253984	6257528	6258700
($n = 32$)	25024136	25027200	25031036
($n = 128$)	100103176	100117420	100157284
bgroup_g2e_equals	3100	3112	3124
bgroup_g2e_hashfromstr_publicinputs (59 bytes)	298524	299884	894696
bgroup_g2e_add_publicinputs	6572	6596	6608
bgroup_g2e_double_publicinputs	2960	2972	2992
bgroup_g2e_negate_publicinputs	60	60	64
bgroup_g2e_scalarmult_publicinputs	612012	625636	635656
bgroup_g2e_scalarmult_base_publicinputs	273468	278372	283056
bgroup_g2e_multiscalarmult_publicinputs			
($n = 2$)	1031736	1043332	1060796
($n = 3$)	1477392	1492796	1510148
($n = 4$)	1889684	1912744	1928124
($n = 8$)	3443640	3467764	3489032
($n = 32$)	10293932	10329088	10366420
($n = 128$)	32941972	32991824	33061804
bgroup_g2e_equals_publicinputs	3104	3116	3120

Table 2. Cycle counts for arithmetic operations in G_2 on Intel Core i5-3210M.

4.2 Implementation with PandA

Our example implementation follows the eBATS API which consists of three functions, namely, `crypto_sign_keypair`, `crypto_sign`, and `crypto_sign_open`. The details of each function are as follows.

The function `crypto_sign_keypair` generates the public and private key pair. It requires one fixed-basepoint scalar multiplication in G_2. The complete code for keypair generation is given in Listing 1. The macro `CRYPTO_BYTES` is required by the eBACS API and is set to `BGROUP_G1E_PACKEDBYTES` in a file called `api.h`.

The function `crypto_sign` computes the signature upon receiving the message. This function also requires hashing to G_1 (we assume that the message is public and use the `publicinputs` version) and one scalar multiplication in G_1. The complete code for signing is given in Listing 2.

The function `crypto_sign_open` verifies whether the signature belongs to the message. As described in the previous subsection, a naive method to compare whether two pairing values are equal is to first compute those two pairings, then compare the results. It is obvious that one can avoid the computation of two pairings. Instead, one computes a product of two pairings and checks whether it is equal to one. In this way, verification needs hashing to G_1 and one pairing-product computation. The code for signature verification is given in Listing 3.

API function	25% quartile	median	75% quartile
bgroup_g3e_unpack	1832068	1832404	1833044
bgroup_g3e_pack	424	424	428
bgroup_g3e_add	8020	8032	8048
bgroup_g3e_double	5548	5560	5572
bgroup_g3e_negate	172	176	180
bgroup_g3e_scalarmult	1120300	1120552	1120936
bgroup_g3e_scalarmult_base	608964	609148	609320
bgroup_g3e_multiscalarmult			
($n = 2$)	2255028	2255624	2258896
($n = 3$)	3382628	3383284	3387392
($n = 4$)	4510336	4511420	4515516
($n = 8$)	9024924	9025736	9026820
($n = 32$)	36100180	36103240	36109596
($n = 128$)	144408660	144446076	144467856
bgroup_g3e_equals	8304	8324	8336
bgroup_g3e_add_publicinputs	8024	8044	8056
bgroup_g3e_double_publicinputs	5548	5556	5568
bgroup_g3e_negate_publicinputs	176	176	180
bgroup_g3e_scalarmult_publicinputs	852272	864136	877804
bgroup_g3e_scalarmult_base_publicinputs	609004	609188	609352
bgroup_g3e_multiscalarmult_publicinputs			
($n = 2$)	2255104	2255424	2258836
($n = 3$)	3382688	3383368	3387800
($n = 4$)	4510680	4512684	4515652
($n = 8$)	4272080	4297668	4330036
($n = 32$)	12768868	12803832	12843124
($n = 128$)	40764052	40825956	40876608
bgroup_g3e_equals_publicinputs	8304	8320	8332

Table 3. Cycle counts for arithmetic operations in G_3 on Intel Core i5-3210M.

4.3 Performance

We benchmarked the BLS implementation on the same Core i5-3210M running at 2.5 GHz that we also used for the detailed benchmarks of our reference implementation of the API. We will also submit the software to eBACS for public benchmarking. Key generation takes 378848 cycles. Signing (of a 59-byte message) takes 434640 cycles (this is a median of 10000 measurements, the quartiles are 428616 and 511764). Verification of a signature on a 59-byte message takes 5832584 cycles (again, this is a median, the quartiles are 5797640 and 5874292). To our knowledge these are the fastest reported speeds of a BLS signature implementation at the 128-bit security level. We would like to compare performance with the BLS implementation by Scott included in SUPERCOP. However, it seems that the software fails to build on 64-bit platforms; consequently eBACS does not contain benchmark results of the "bls" software on such platforms.

We ran the benchmark included in the RELIC framework (version 0.3.5) on the same machine that we used for benchmarking. The times reported by this RELIC benchmark are 609966 nanoseconds for BLS key generation, 510775 nanoseconds for signing and 6910615 nanoseconds for verification. At a clock speed of 2.5 GHz this corresponds to 1524915 cycles for key generation, 1276937 cycles for signing, and 17276537 cyles for verification; about three times slower than our implementation.

API function	25% quartile	median	75% quartile
bgroup_pairing	2566580	2567116	2572096
bgroup_pairing_product			
$(n = 2)$	3831724	3832644	3837688
$(n = 3)$	5089192	5093724	5094728
$(n = 4)$	6347328	6351260	6352588
$(n = 8)$	11380604	11381384	11383420
$(n = 32)$	41565448	41569424	41588976
$(n = 128)$	162321836	162364916	162387468

Table 4. Cycle counts for pairing computation on Intel Core i5-3210M.

Listing 1 Public and private key generation

```
int crypto_sign_keypair(
  unsigned char *pk,
  unsigned char *sk
)
{
  // private key //
  bgroup_scalar x;
  bgroup_scalar_setrandom(&x);
  bgroup_scalar_pack(sk, &x);

  // public key //
  bgroup_g2e r;
  bgroup_g2_scalarmult_base(&r, &x);
  bgroup_g2_pack(pk, &r);

  return 0;
}
```

Listing 2 Signature generation

```
int crypto_sign(
  unsigned char *sm,
  unsigned long long *smlen,
  const unsigned char *m,
  unsigned long long mlen,
  const unsigned char *sk)
{

  bgroup_g1e p, p1;
  bgroup_scalar x;
  int i,r;

  bgroup_g1e_hashfromstr_publicinputs(&p, m, mlen);
  r = bgroup_scalar_unpack(&x, sk);
  bgroup_g1e_scalarmult(&p1, &p, &x);
  bgroup_g1e_pack(sm, &p1);

  for (i = 0; i < mlen; i++)
  sm[i + CRYPTO_BYTES] = m[i];
  *smlen = mlen + CRYPTO_BYTES;

  return -r;
}
```

Listing 3 Signature verification

```
int crypto_sign_open(
  unsigned char *m,
  unsigned long long *mlen,
  const unsigned char *sm,
  unsigned long long smlen,
  const unsigned char *pk)
{

  bgroup_g1e p[2];
  bgroup_g2e q[2];
  bgroup_g3e r;
  unsigned long long i;
  int ok;

  ok  = !bgroup_g1e_unpack(p, sm);
  bgroup_g1e_negate_publicinputs(p, p);
  q[0] = bgroup_g2e_base;
  bgroup_g1e_hashfromstr_publicinputs(p+1, sm + CRYPTO_BYTES, smlen - CRYPTO_BYTES);
  ok &= !bgroup_g2e_unpack(q+1, pk);
  bgroup_pairing_product(&r, p, q, 2);

  ok &= bgroup_g3e_equals(&r, &bgroup_g3e_neutral);

  if (ok)
  {
    for (i = 0; i < smlen - CRYPTO_BYTES; i++)
      m[i] = sm[i + CRYPTO_BYTES];
    *mlen = smlen - CRYPTO_BYTES;
    return 0;
  }
  else
  {
    for (i = 0; i < smlen - CRYPTO_BYTES; i++)
      m[i] = 0;
    *mlen = (unsigned long long) (-1);
    return -1;
  }
}
```

References

1. Gora Adj, Alfred Menezes, Thomaz Oliveira, and Francisco Rodríguez-Henríquez. Weakness of $\mathbb{F}_{3^{6\cdot 509}}$ for discrete logarithm cryptography. Cryptology ePrint Archive, Report 2013/446, 2013. http://eprint.iacr.org/2013/446/. 2
2. Gora Adj and Francisco Rodríguez-Henríquez. Square root computation over even extension fields. Cryptology ePrint Archive, Report 2012/685, 2012. http://eprint.iacr.org/. 8
3. Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, 2013. http://eprint.iacr.org/2011/617/. 4
4. D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. http://code.google.com/p/relic-toolkit/. 4
5. Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio López. Faster explicit formulas for computing pairings over ordinary curves. In Kenneth G. Paterson, editor, *Advances in Cryptology – Eurocrypt 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 48–68. Springer, 2011. http://eprint.iacr.org/2010/526/. 2, 8, 10
6. Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. Cryptology ePrint Archive, Report 2013/400, 2013. http://eprint.iacr.org/2013/400/. 2
7. Paulo S.L.M. Barreto, H. Y. Kim, B. Lynn, and M. Scott. Efficient algorithms for pairing-based cryptosystems. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2002. http://eprint.iacr.org/2002/008. 1
8. Paulo S.L.M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer, 2006. http://cryptosith.org/papers/#bn. 8
9. Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1987. 10
10. Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. Randomizable proofs and delegatable anonymous credentials. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2009. http://research.microsoft.com/pubs/122759/anoncred.pdf. 1
11. Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards curves. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer, 2008. http://cr.yp.to/papers.html#twisted. 3
12. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011. see also full version [13]. 3, 9, 16
13. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. http://cryptojedi.org/papers/#ed25519, see also short version [12]. 3, 9, 16
14. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to (accessed 2013-08-15). 2, 10
15. Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, 2007. http://cr.yp.to/papers.html#newelliptic. 3
16. Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2012. http://cryptojedi.org/papers/#coolnacl. 4
17. Jean-Luc Beuchat, Jorge E. González Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves, 2010. http://eprint.iacr.org/2010/354/. 2, 10
18. Dan Boneh and Matt Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2001. http://www.iacr.org/archive/crypto2001/21390212.pdf. 1
19. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004. http://crypto.stanford.edu/~dabo/pubs/papers/weilsigs.ps. 1, 9, 10
20. Joppe W. Bos, Craig Costello, and Michael Naehrig. Exponentiating in pairing groups. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography – SAC 2013*, volume to appear of *Lecture Notes in Computer Science*, 2013. http://cryptosith.org/papers/#exppair. 2, 9, 10

21. Wieb Bosma and Hendrik W. Lenstra. Complete systems of two addition laws for elliptic curves. *Journal of Number Theory*, 53:229–240, 1995. http://www.math.ru.nl/~bosma/pubs/JNT1995.pdf. 9
22. Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security—ESORICS 2011*, volume 6879 of *LNCS*, pages 355–371. Springer, 2011. http://eprint.iacr.org/2011/232/. 3
23. Certivox. MIRACL Cryptographic SDK. http://www.certivox.com/miracl. 4
24. Sanjit Chatterjee and Alfred Menezes. On cryptographic protocols employing asymmetric pairings – the role of ψ revisited. *Discrete Applied Mathematics*, 159:1311–1322, 2011. http://eprint.iacr.org/2009/480/. 3
25. Peter de Rooij. Efficient exponentiation using precomputation and vector addition chains. In Alfredo De Santis, editor, *Advances in Cryptology – EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*, pages 389–399. Springer, 1995. 10
26. Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE Computer Society, 2013. www.isg.rhul.ac.uk/tls/TLStiming.pdf. 3
27. Pierre-Alain Fouque and Mehdi Tibouchi. Indifferentiable hashing to Barreto–Naehrig curves. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2012. www.di.ens.fr/~fouque/pub/latincrypt12.pdf. 9
28. David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23(2):224–280, 2010. http://eprint.iacr.org/2006/372/. 1
29. Craig Gentry and Alice Silverberg. Hierarchical id-based cryptography. In Yuliang Zheng, editor, *Advances in Cryptology – ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 548–566. Springer, 2002. http://www.cs.ucdavis.edu/~franklin/ecs228/pubs/extra_pubs/hibe.pdf. 1
30. The GNU MP library. http://gmplib.org/ (accessed 2013-11-02). 6
31. Faruk Göloğlu, Robert Granger, Gary McGuire, and Jens Zumbrägel. Solving a 6120-bit DLP on a desktop computer. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography*, volume to appear of *Lecture Notes in Computer Science*. Springer, 2013. http://eprint.iacr.org/2013/306. 2
32. Robert Granger, Dan Page, and Martijn Stam. On small characteristic algebraic tori in pairing-based cryptography. *IACR Cryptology ePrint Archive*, page 132, 2004. http://eprint.iacr.org/2004/132. 8
33. Robert Granger and Michael Scott. Faster squaring in the cyclotomic subgroup of sixth degree extensions. In Phong. Q. Nguyen and David Pointcheval, editors, *Public Key Cryptography – PKC 2010*, volume 6056 of *LNCS*, pages 209–223. Springer, 2010. 10
34. Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *Advances in Cryptology – ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2010. http://www.cs.ucl.ac.uk/staff/J.Groth/ShortNIZK.pdf. 1
35. Jens Groth and Amit Sahai. Efficient noninteractive proof systems for bilinear groups. *SIAM J. Comput.*, 41(5):1193–1232, 2012. http://www0.cs.ucl.ac.uk/staff/J.Groth/WImoduleFull.pdf. 1
36. F. Hess, N. P. Smart, and F. Vercauteren. The eta pairing revisited. *IEEE Transactions on Information Theory*, 52(10):4595–4602, 2006. http://eprint.iacr.org/2006/110. 2
37. Hüseyin Hışıl. *Elliptic Curves, Group Law, and Efficient Computation*. PhD thesis, Queensland University of Technology, 2010. http://eprints.qut.edu.au/33233/. 9
38. Jeremy Horwitz and Ben Lynn. Toward hierarchical identity-based encryption. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 466–481. Springer, 2002. http://theory.stanford.edu/~horwitz/pubs/hibe.pdf. 1
39. Antoine Joux. A one round protocol for tripartite Diffie-Hellman. In Wieb Bosma, editor, *Algorithmic Number Theory*, volume 1838 of *Lecture Notes in Computer Science*, pages 385–393. Springer, 2000. cgi.di.uoa.gr/~aggelos/crypto/page4/assets/joux-tripartite.pdf. 1
40. Antoine Joux. A one round protocol for tripartite Diffie-Hellman. *Journal of Cryptology*, 17(4):263–276, 2004. 1
41. Antoine Joux. A new index calculus algorithm with complexity $L(1/4+o(1))$ in very small characteristic. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography*, volume to appear of *Lecture Notes in Computer Science*. Springer, 2013. invited paper, http://eprint.iacr.org/2013/095/. 2
42. Ben Lynn. PBC library – the pairing-based cryptography library. http://crypto.stanford.edu/pbc/. 4
43. Shigeo MITSUNARI. A fast implementation of the optimal ate pairing over BN curve on Intel Haswell processor. Cryptology ePrint Archive, Report 2013/362, 2013. http://eprint.iacr.org/2013/362/. 2
44. Michael Naehrig, Paulo S.L.M. Barreto, and Peter Schwabe. On compressible pairings and their computation. In Serge Vaudenay, editor, *Progress in Cryptology - AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2008. http://eprint.iacr.org/2007/429/. 8
45. Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In Michel Abdalla and Paulo S.L.M. Barreto, editors, *Progress in Cryptology – LATINCRYPT*

2010, volume 6212 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2010. updated version: http://cryptojedi.org/users/peter/#dclxvi. 2, 10
46. Kiyoshi Ohgishi, Ryuichi Sakai, and Masao Kasahara. Notes on ID-based key sharing systems over elliptic curve (in Japanese). Technical Report ISEC99-57, IEICE, 1999. 1
47. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. http://eprint.iacr.org/2005/271/. 3
48. Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In IEEE, editor, *Proceedings of the IEEE Symposium on Security and Privacy*, 2013. http://eprint.iacr.org/2013/279. 2
49. Geovandro C.C.F. Pereira, Marcos A. Simplício Jr, Michael Naehrig, and Paulo S.L.M. Barreto. A family of implementation-friendly BN elliptic curves. *Journal of Systems and Software*, 84(8):1319–1326, 2011. http://cryptojedi.org/papers/#fast-bn. 8
50. Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 457–473. Springer, 2005. http://eprint.iacr.org/2004/086/. 1
51. Ryuichi Sakai, Kiyoshi Ohgishi, and Masao Kasahara. Cryptosystems based on pairing. In *The 2000 Symposium on Cryptography and Information Security, Okinawa, Japan*, pages 135–148, 2000. 1
52. Ryuichi Sakai, Kiyoshi Ohgishi, and Masao Kasahara. Cryptosystems based on pairing over elliptic curve (in Japanese). In *The 2001 Symposium on Cryptography and Information Security, Oiso, Japan*, pages 23–26, 2001. 1
53. Ana Helena Sánchez and Francisco Rodríguez-Henríquez. NEON implementation of an attribute-based encryption scheme. In Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security*, volume 7954 of *Lecture Notes in Computer Science*, pages 322–338. Springer, 2013. http://cacr.uwaterloo.ca/techreports/2013/cacr2013-07.pdf. 2
54. Michael Scott. On the efficient implementation of pairing-based protocols. In Liqun Chen, editor, *Cryptography and Coding*, volume 7089 of *Lecture Notes in Computer Science*, pages 296–308. Springer, 2011. http://eprint.iacr.org/2011/334/. 2
55. Michael Scott and Paulo S.L.M. Barreto. Compressed pairings. In Matthew K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2004. 8
56. Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J. Kachisa. On the final exponentiation for calculating pairings on ordinary elliptic curves. In Hovav Shacham and Brent Waters, editors, *Pairing-Based Cryptography – Pairing 2009*, volume 5671 of *Lecture Notes in Computer Science*, pages 78–88. Springer, 2009. eprint.iacr.org/2008/490/. 2
57. Andrew Shallue and Christiaan E. van de Woestijne. Construction of rational points on elliptic curves over finite fields. In *Proceedings of the 7th international conference on Algorithmic Number Theory*, ANTS'06, pages 510–524, Berlin, Heidelberg, 2006. Springer-Verlag. 9
58. Martijn Stam and Arjen K. Lenstra. Efficient subgroup exponentiation in quadratic and sixth degree extensions. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *LNCS*, pages 318–332. Springer, 2003. 10
59. Tadanori Teruya, Kazutaka Saito, Naoki Kanayama, Yuto Kawahara, Tetsutaro Kobayashi, and Eiji Okamoto. Constructing symmetric pairings over supersingular elliptic curves with embedding degree three. In Zhenfu Cao and Fangguo Zhang, editors, *Pairing-Based Cryptography – Pairing 2013*, Lecture Notes in Computer Science. Springer, to appear. 3
60. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010. http://people.csail.mit.edu/tromer/papers/cache-joc-official.pdf. 3
61. Frederik Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56(1), 2010. http://www.cosic.esat.kuleuven.be/publications/article-1039.pdf. 2
62. Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, L3 cache side-channel attack. Cryptology ePrint Archive, Report 2013/448, 2013. http://eprint.iacr.org/2013/448/. 3
63. Xusheng Zhang and Kunpeng Wang. Fast symmetric pairing revisited. In Zhenfu Cao and Fangguo Zhang, editors, *Pairing-Based Cryptography – Pairing 2013*, Lecture Notes in Computer Science. Springer, to appear. 3